5. 오차역전파법

신경망의 가중치 매개변수에 대한 손실 함수의 기울기는 수치 미분을 사용해 구했다. 수치미분은 단순하고 구현도 쉽지만 계산 시간이 오래 걸린다.

오차역전파법은 가중치 매개변수의 기울기를 효율적으로 계산한다.

5.1 계산 그래프

5.1.1 계산 그래프로 풀다

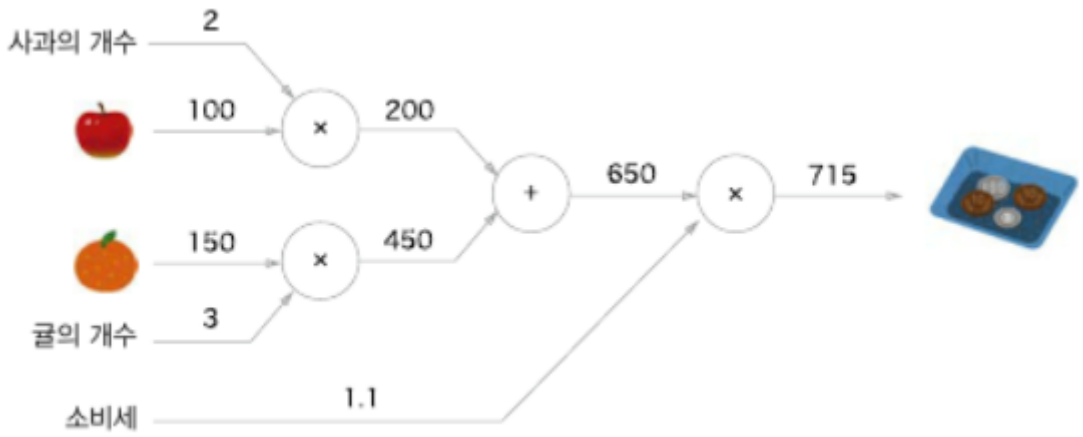
계산 그래프는 그래프 자료구조로 계산 과정을 그래프(노드, 엣지)로 나타낸 것이다.

노드(원)에는 연산 내용을 적고, 변수는 원 밖에 화살표로 표시하고, 화살표 위에는 계산 결과를 적는다.

계산 그래프의 계산 흐름은 왼쪽에서 오른쪽으로 진행한다.

국소적(자신과 직접 관계된 작은 범위) 계산을 전파함으로써 최종 결과를 얻는다.

문제: 1개에 100원짜리 사과 2개, 1개에 150원짜리 귤 3개를 구매하려 한다. 소비세가 10%일 때 지불 금액을 구하시오



100 * 2 = 200

150 * 3 = 450

200 + 450 = 650

650 * 1.1 = 715

5.1.2 국소적 계산

계산 그래프는 국소적 계산에 집중한다. 국소적 계산은 결국 전체에서 어떤 일이 벌어지든 상관없이 자신과 관계된 정보만으로 결과를 출력할 수 있다는 것이다.

전체 계산이 아무리 복잡하더라도 각 단계에서 하는 일은 해당 노드의 국소적 계산이다. 국소적 계산의 결과를 전달함으로써 전체를 구성하는 복잡한 계산을 해낼 수 있다.

5.1.3 왜 계산 그래프로 푸는가?

실제 계산 그래프를 사용하는 가장 큰 이유는 역전파를 통해 미분을 효율적으로 계산할 수 있는 점에 있다.

역전파는 반대 방향의 굵은 화살표로 그린다. 역전파는 국소적 미분을 전달하고 그 미분 값은 화살표의 아래에 적는다.

계산 그래프의 이점은 순전파와 역전파를 활용해서 각 변수의 미분을 효율적으로 구할 수 있다는 것이다.

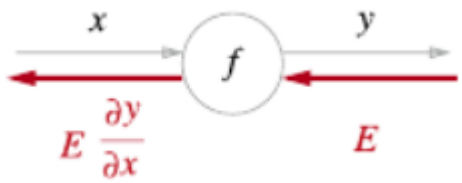
5.2 연쇄법칙

연쇄법칙: 국소적 미분을 전달하는 원리. 계산 그래프 상의 역전파와 같다는 사실을 밝히겠다.

5.2.1 계산 그래프의 역전파

`#역전파` `#효율적미분계산` `#연쇄법칙`

y=f(x)의 계산 그래프:



국소적 미분: 순전파 때의 y=f(x) 계산의 미분(x에 대한 y의 미분)을 구함

국소적인 미분을 상류에서 전달된 값에 곱해 앞쪽 노드로 전달한다. 이러한 방식을 따르면 목표로 하는 미분 값을 효율적으로 구할 수 있다는 것이 역전파의 핵심이다. 연쇄법칙의 원리로 설명 가능하다.

5.2.2 연쇄법칙이란?

`#연쇄법칙`

합성 함수: 여러 함수로 구성된 함수

연쇄법칙: 연쇄법칙은 `합성 함수의 미분`에 대한 성질이며, 합성 함수의 미분은 `합성 함수를 구성하는 각 함수의 미분의 곱`으로 나타낼 수 있다.

z = t^2

t = x + y

이 때, dz/dx (x 에 대한 z의 미분) 는 (dz/dt) * (dt/dx) 으로 나타낼 수 있다는 것이다.

연쇄 법칙을 써서 dz/dx를 구해보자. 가장 먼저 국소적 미분(편미분)을 구한다.

dz/dt = 2t

dt/dx = 1

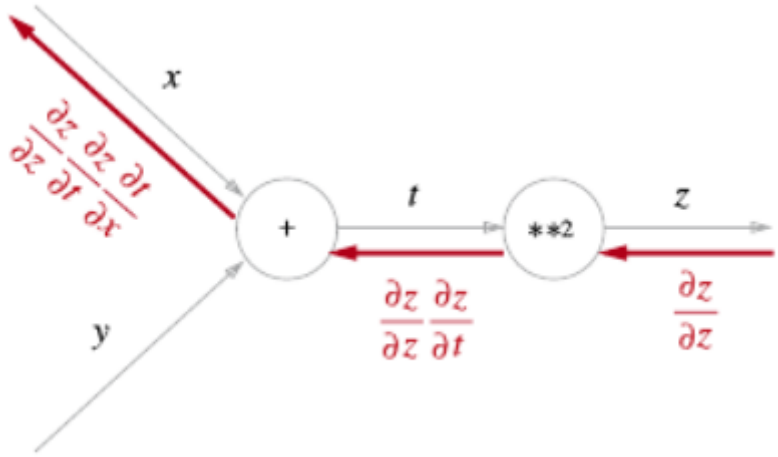
수치적 미분이 아닌 해석적 미분으로 구한 결과이다.

최종적으로 구하고 싶은 dz/dx는 dz/dt 와 dt/dx 두 미분을 곱해 계산한다.

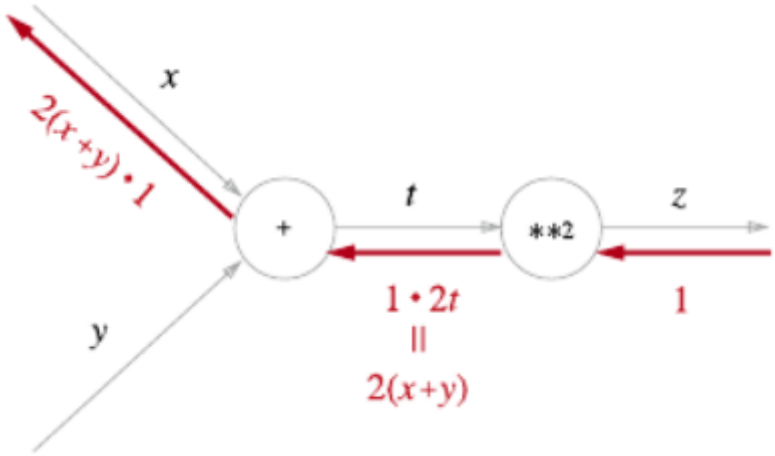
dz/dx = (dz/dt) * (dt/dx) = 2t * 1 = 2(x+y)

5.2.3 연쇄법칙과 계산 그래프

#계산그래프로 표현



맨 왼쪽 역전파는 (dz/dz) * (dz/dt) * (dt/dx) = (dz/dt) * (dt/dx) = (dz/dx) 가 성립되어 x에 대한 z의 미분이 된다. 즉, 역전파가 하는 일은 연쇄법칙의 원리와 같다.



연쇄법칙의 결과와 동일하게 2(x+y) 가 나온다.

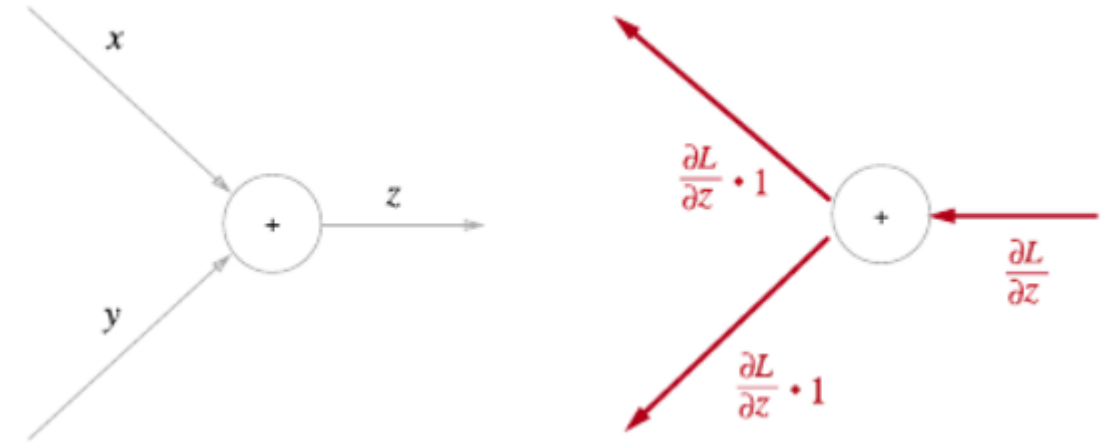
5.3 역전파

#역전파의 구조 설명

5.3.1 덧셈 노드의 역전파

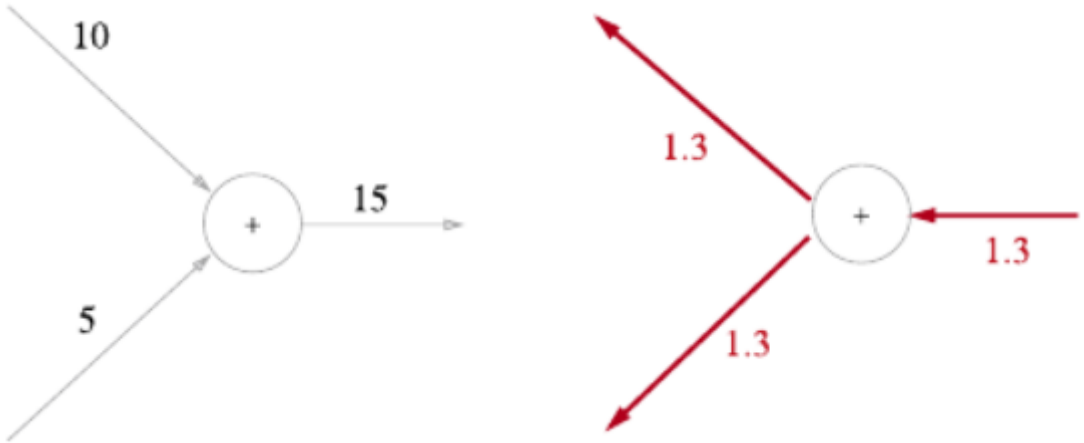
z = x + y

미분(해석적): dz/dx = 1 , dz/dy = 1



순전파 역전파

덧셈 노드의 역전파는 1을 곱하기만 할 뿐이므로 입력된 값을 그대로 다음 노드로 보내게 된다.



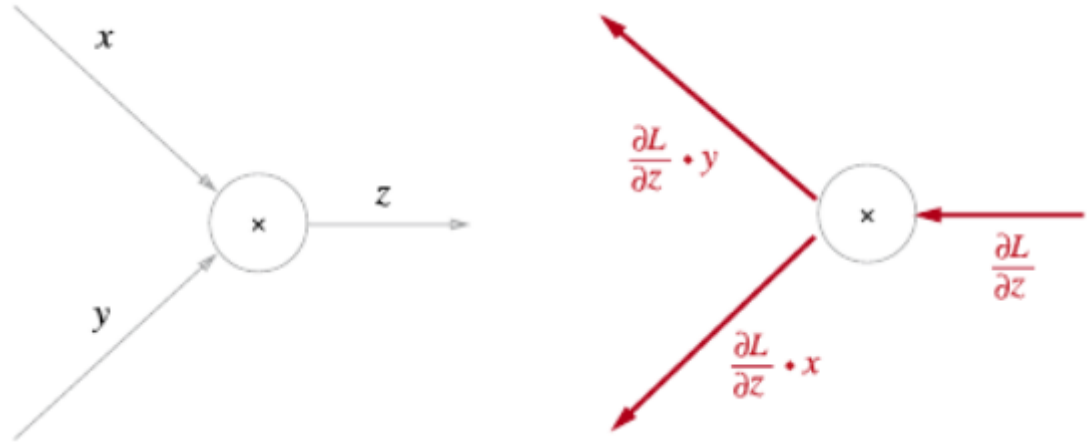
10 + 5 = 15 라는 계산이 있고, 역전파 시 상류에서 1.3이라는 값이 흘러온다.

덧셈 노드 역전파는 입력 신호를 다음 노드로 출력할 뿐이므로 1.3을 그대로 다음 노드로 전달한다.

5.3.2 곱셈 노드의 역전파

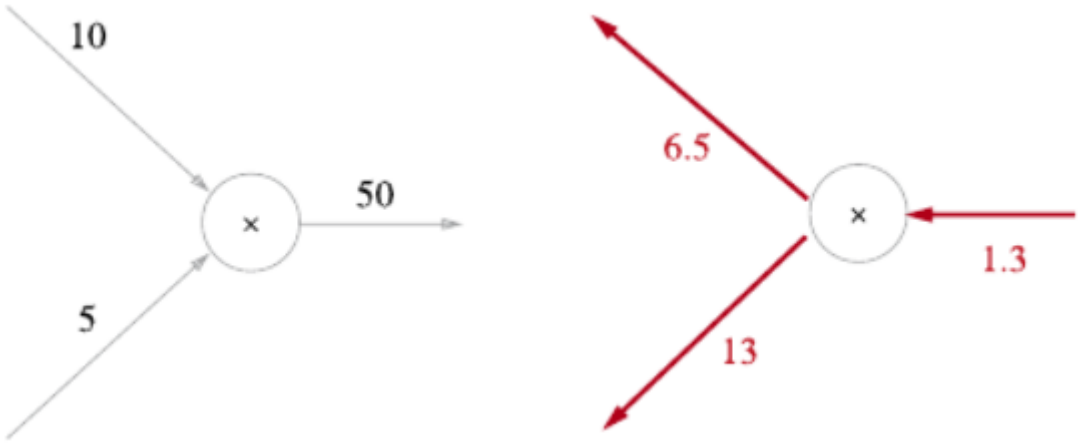
z= x* y

미분(해석적) : dz/dx = y , dz/dy = x



순전파 역전파

곱셈 노드 역전파는 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보낸다.

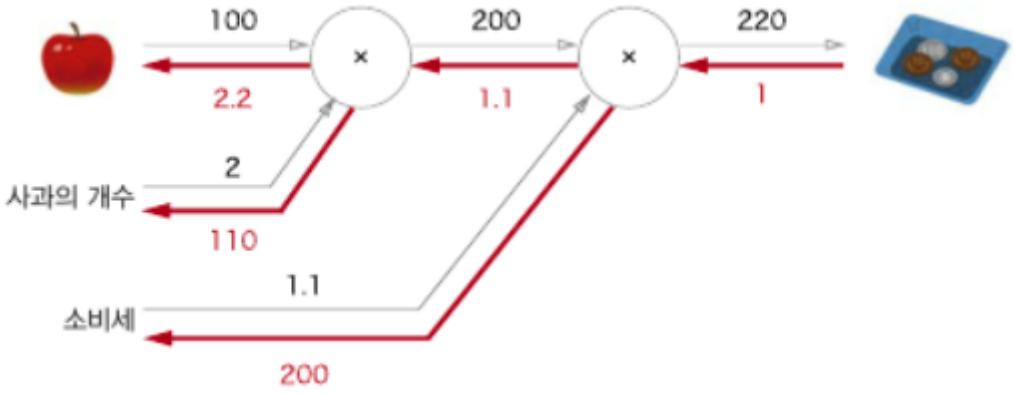


10 x 5 = 50 라는 계산이 있고, 역전파 시 상류에서 1.3이라는 값이 흘러온다.

곱셈 노드 역전파는 상류의 값에 순방향 입력 신호를 바꾼 값을 곱하여 출력한다. 그래서 순방향 입력 신호의 값이 필요하며 곱셈 노드를 구현할 때는 순전파의 입력 신호를 변수에 저장해둔다.

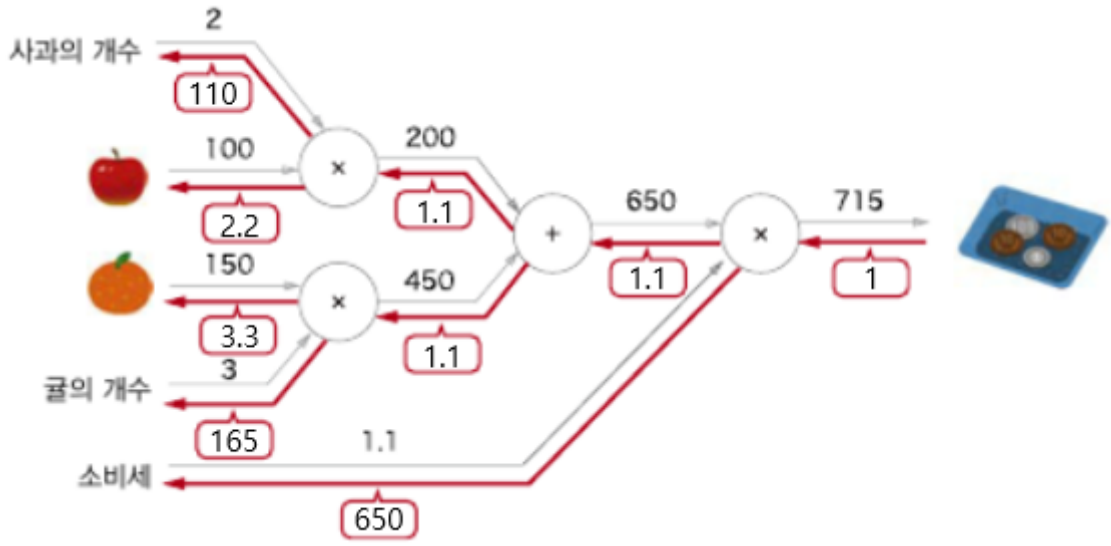
5.3.3 사과 쇼핑의 예

변수: 사과의 가격, 사과의 개수, 소비세



미분의 결과는 해당 변수가 1단위만큼 바뀐다면 결과값에 영향을 주는 정도로 해석할 수 있다.

문제:



5.4 단순한 계층 구현하기

신경망을 구성하는 계층(신경망의 기능 단위) 각각을 하나의 클래스로 구현해보자.

모든 계층은 forward()와 backward()라는 공통의 메서드(인터페이스)를 갖도록 구현

덧셈 노드를 AddLayer, 곱셈 노드를 MunLayer 로 구현해보자.

5.4.1 곱셈 계층

```
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y

        return out

    def backward(self, dout):
        dx = dout * self.y
        dy = dout * self.x

        return dx, dy
```

`__init__()` 에서는 인스턴수 변수(x, y)를 초기화한다. 이 두 변수는 순전파 시 입력 값을 유지하기 위해서 사용한다.

`forward()` 에서는 x와 y를 인수로 받고 두 값을 곱해서 반환한다.

`backward()` 에서는 상류에서 넘어온 미분(dout)에 순전파 때의 값을 서로 바꿔 곱한 후 하류로 흘린다.

```
# 예시
apple = 100
apple_num = 2
tax = 1.1

# 계층들
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

# 순전파
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)

print(price)

# 역전파
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print(dapple, dapple_num, dtax)
```

`backward()`가 받는 인수는 '순전파의 출력에 대한 미분'임에 주의하자.

즉, `forward()`의 출력값을 미분하여 `backward()`에 넣어야 한다.

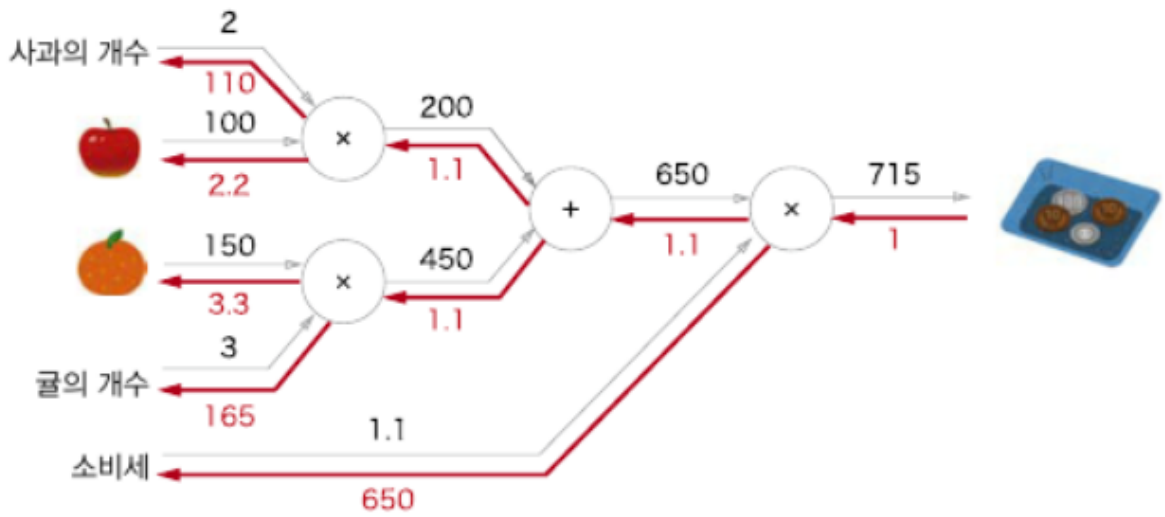
5.4.2 덧셈 계층

```
class AddLayer:
    def __init__(self):
        pass

    def forward(self, x, y):
        out = x + y
        return out

    def backward(self, dout):
        dx = dout * 1
        dy = dout*1
        return dx, dy
```

`__init__()`에서는 아무일도 하지 않는다. 덧셈 계층에서는 초기화가 필요 없기 때문이다.

`forward()`에서는 입력받은 두 인수 x, y를 더해서 반환한다.

`backward()`에서는 상류에서 내려온 미분(dout)을 그대로 하류로 흘린다.




```
apple, apple_num = 100, 2
orange, orange_num = 150, 3
tax = 1.1

# 계층들
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# 순전파
```

```
apple_price = mul_apple_layer.forward(apple, apple_num)
orange_price = mul_orange_layer.forward(orange, orange_num)
all_price = add_apple_orange_layer.forward(apple_price, orange_price)
price = mul_tax_layer.forward(all_price, tax)

# 역전파
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice)
dapple_price, dorange_price = add_apple_orange_layer.packward(dall_price)
dorange, dorange_num = mul_orange_layer.backward(dorange_price)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print(price)
```

5.5 활성화 함수 계층 구현하기

`#계산 그래프를 신경망에 적용`

신경망을 구성하는 층(계층) 각각을 클래스 하나로 구현한다.

ReLU와 Sigmoid 계층을 구현해보자.

5.5.1 ReLU 계층

ReLU 함수: $y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$ 미분: $\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}$

즉, 순전파 때의 입력인 x가 0보다 트면 역전파는 상류의 값을 그대로 하류로 흘린다. 하지만 순전파 때 x가 0 이하면 역전파 대는 하류로 신호를 보내지 않는다.


```
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout
```

```
        return dx
```

forward()와 backward()함수는 넘파이 배열을 인수로 받는다고 가정한다.

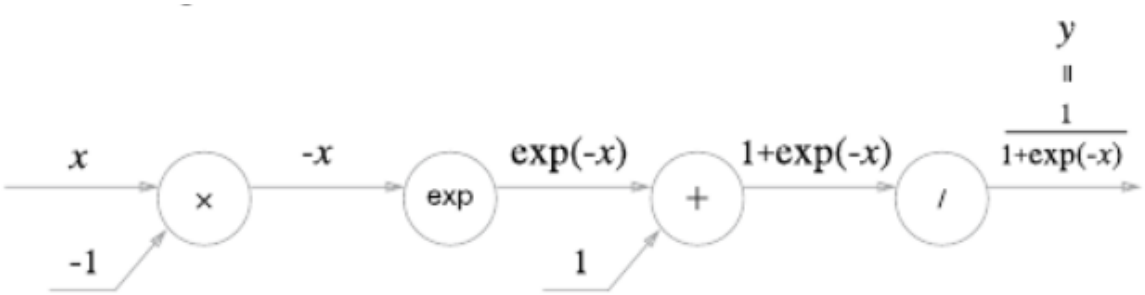
mask 는 인스턴스 변수로 True/False로 구성된 넘파이 배열이다. 순전파이— 입력인 x의 원소 값이 0 이하인 인덱스는 True, 0보다 큰 원소는 False로 유지한다.

역전파 때는 순전파 때 만들어둔 mask를 써서 True인 곳(0이하)에는 상류에서 전파된 dout를 0으로 설정한다.

5.5.2 Sigmoid 계층

시그모이드 함수: $y = \frac{1}{1 + \exp(-x)}$

계산 그래프 (순전파):



`exp` 노드는 y = exp(x) 계산을 수행한다.

`/` 노드는 y = 1/x 계산을 수행한다.

역전파의 흐름을 오른쪽에서 왼쪽으로 한 단계씩 짚어보자.

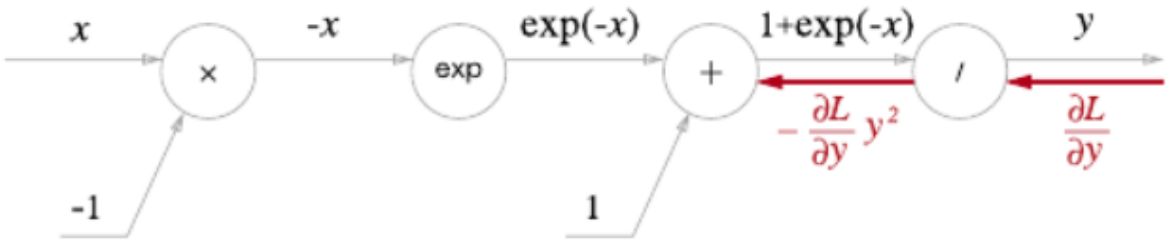
1단계

`/` 노드(y = 1/x) 를 미분하면

$$\frac{\partial y}{\partial x} = -\frac{1}{x^2}$$

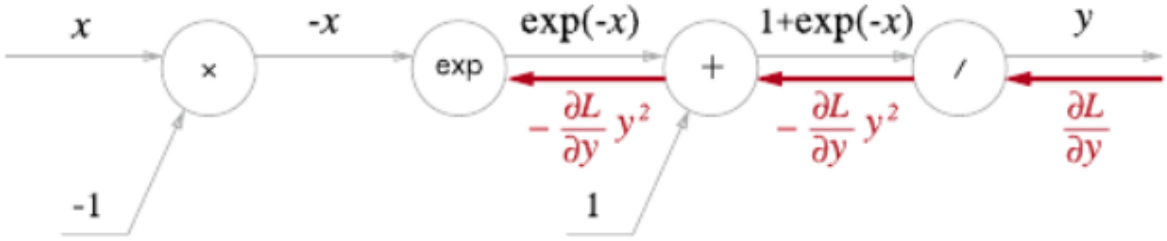
$$= -y^2$$

가 된다. 즉, 역전파 때 상류에서 흘러온 값에 `순전파의 출력을 제곱한 후 마이너스를 붙인 값`을 곱해서 하류로 전달한다.



2단계

`+`노드는 상류의 값을 여과없이 하류로 내보낸다.

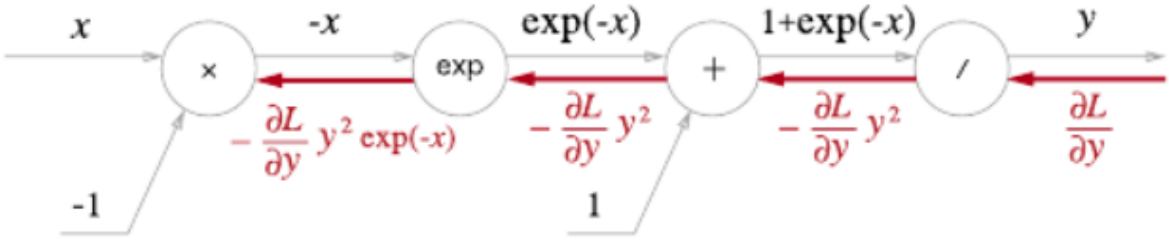


3단계

`exp`노드는 y= exp(x) 연산을 수행한다.

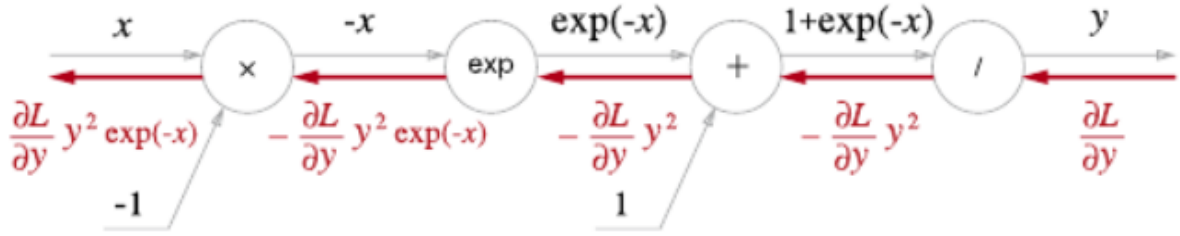
exp(x) 미분 값: $\frac{\partial y}{\partial x} = \exp(x)$

상류의 값에 순전파 때의 출력을 곱해 하류로 전파한다.



4단계

`x`노드는 순전파 때의 값을 서로 바꿔 곱한다.



최종 결과인 $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 값은 순전파의 입력 x와 출력 y만으로 계산할 수 있다. 그래서 계산 그래프의 중간 과정을 모두 묶어 단순한 sigmoid 노드 하나로 대체할 수 있다.

계산 그래프와 간소화 버전의 결과는 같지만 간소화 버전은 역전파 과정의 중간 계산들을 생략하여 더 효율적인 계산이다.

$$\frac{\partial L}{\partial y} y^2 \exp(-x) = \frac{\partial L}{\partial y} \frac{1}{(1+\exp(-x))^2} \exp(-x)$$

또한

$$= \frac{\partial L}{\partial y} \frac{1}{1+\exp(-x)} \frac{\exp(-x)}{1+\exp(-x)}$$

$$= \frac{\partial L}{\partial y} y(1-y)$$

를 통하여 시그모이드 계층의 역전파는 순전파의 출력(y)만으로 계산할 수 있다는 것을 알 수 있다.

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out

        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

이 구현에서는 순전파의 출력을 인스턴스 변수 out에 보관했다가 역전파 계산 때 사용한다.

5.6 Affine/Softmax 계층 구현하기

#Softmax #CrossEntropyLoss

5.6.1 Affine 계층

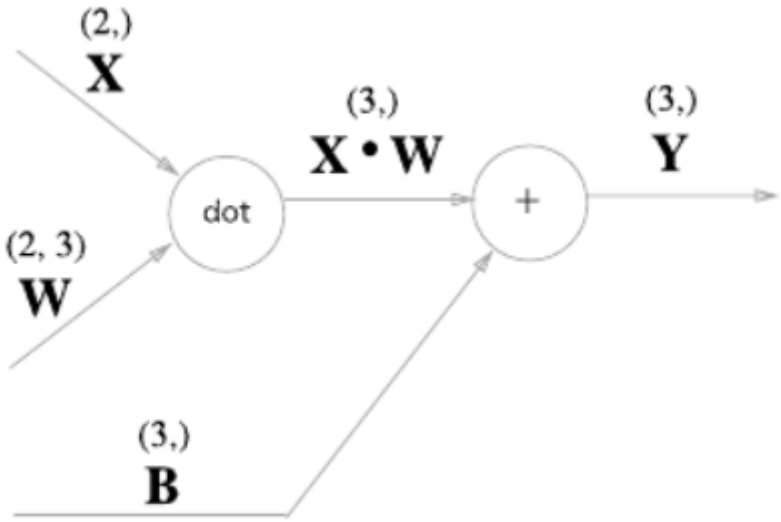
신경망의 순전파 때 수행하는 행렬의 곱은 기하학에서는 어파인 변환(affine transformation)이라고 한다.

신경망의 순전파에서는 가중치 신호의 총합을 계산하기 때문에 행렬의 곱(np.dot())을 사용했다.

행렬의 곱 계산은 대응하는 차원의 원소 수를 일치시키는게 핵심이다.

행렬의 곱과 편향의 합을 계산그래프로 그려보자.

곱 계산 노드를 `dot`라고 하면 np.dot(X,W) + B 계산은 다음과 같다.



행렬을 사용한 역전파도 행렬의 원소마다 전개해보면 스칼라 값을 사용한 지금까지의 계산 그래프와 같은 순서로 생각할 수 있다.

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^{\mathrm{T}}$$

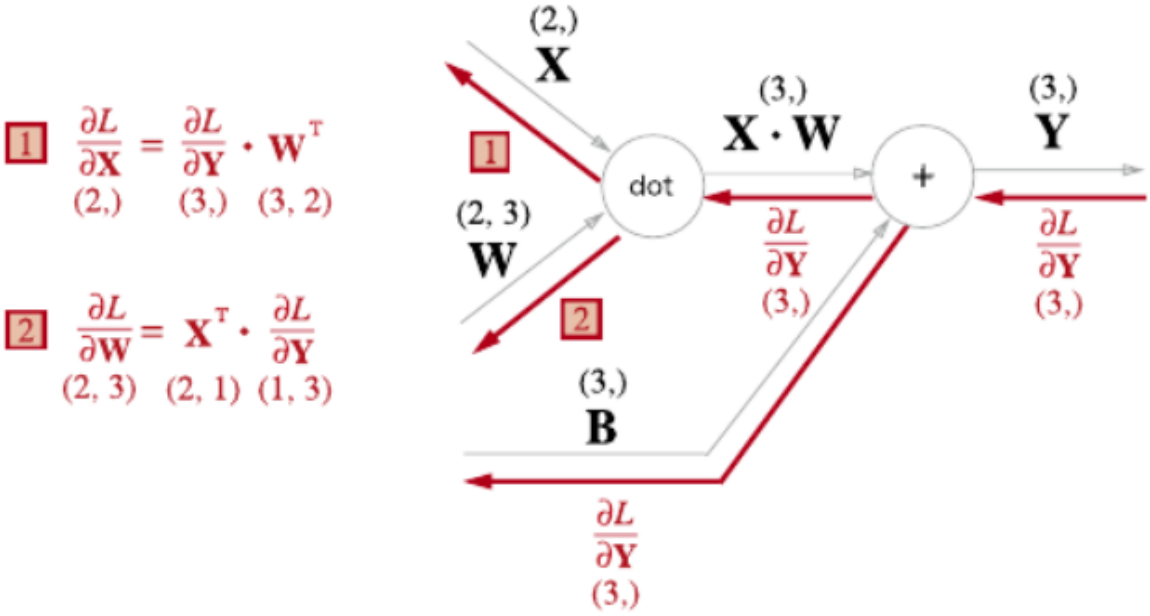
$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{Y}}$$

W^T 의 ^T는 전치행렬을 뜻한다.

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\mathbf{W}^{\mathrm{T}} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix}$$

역전파 계산 흐름은 다음과 같다.



행렬의 형상에 주의해야 한다. 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시켜야한다. X와 dL/dX는 같은 형상이고, W와 dL/dW도 같은 형상이다.

$$\mathbf{X} = (x_0, x_1, \cdots, x_n)$$

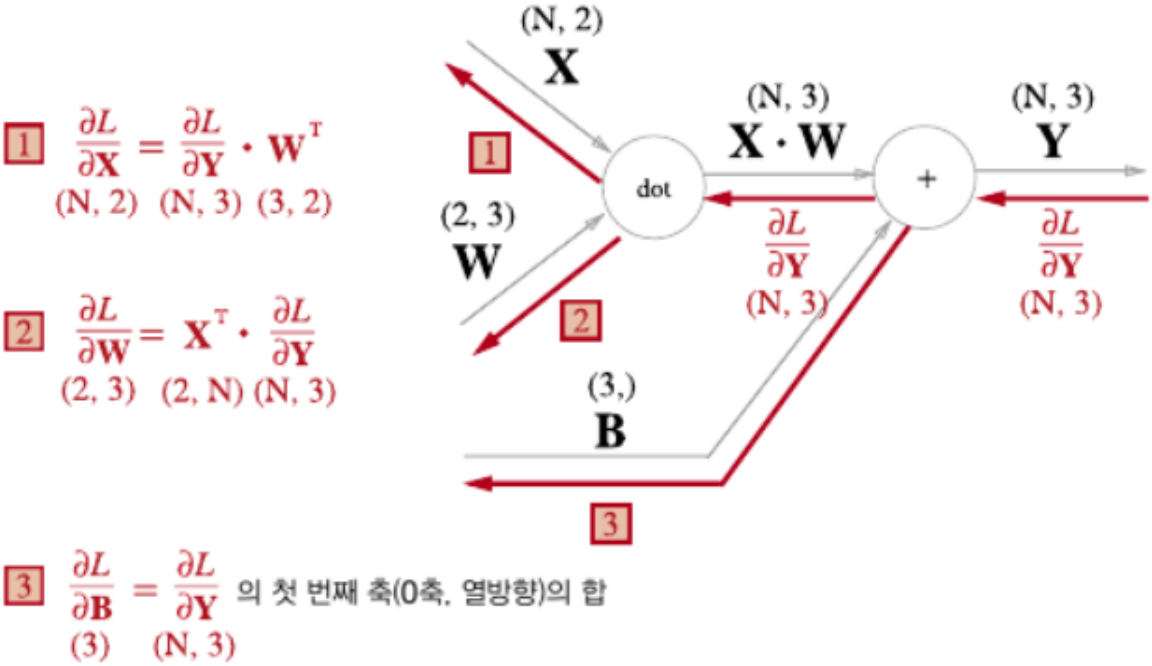
$$\frac{\partial L}{\partial \mathbf{X}} = \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \cdots, \frac{\partial L}{\partial x_n} \right)$$

행렬 곱의 역전파는 행렬의 대응하는 차원의 원소 수가 일치하도록 곱을 조립하여 구할 수 있다.

5.6.2 배치용 Affine 계층

지금까지는 입력 데이터로 행렬 X 하나만을 고려한 것이다.

데이터 N개를 묶어 순전파하는 경우(배치용 Affine 계층)를 생각해보자.



입력 X의 형상이 (2,) 에서 (N, 2)가 되었다. 편향은 데이터 각각에 더해진다. 그래서 역전파 때는 각 데이터의 역전파 값이 편향의 원소에 모여야 한다. (np.sum(dY, axis=0))

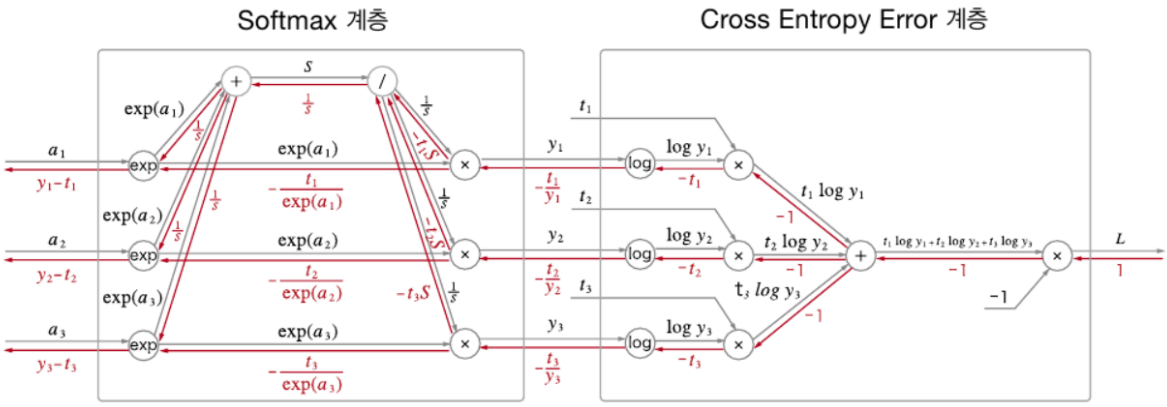
Affine 구현

```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.x = None
        self.dW = None
        self.db = None

    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b

        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)    # .T는 전치행렬
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)
```

5.6.3 Softmax-with-Loss 계층

출력층에서 사용하는 소프트 맥스 함수는 입력 값을 정규화(출력의 합이 1이 되도록 변형)하여 출력한다.

손실 함수인 교차 엔트로피 오차도 포함하여 Softmax-withLoss 계층을 구현해보자.



Softmax 계층은 입력 (a1, a2, a3)를 정규화하여 (y1, y2, y3)를 출력한다.

Cross Entropy Error 계층은 Softmax의 출력(y1, y2, y3)와 정답 레이블 (t1, t2, t3)를 받고, 이 데이터들로부터 손실 L을 출력한다.

역전파의 결과는 (y1-t1, y2-t2, y3-t3) 이다. 신경망의 역전파에서는 이 차이인 오차가 앞 계층에 전해지는 것이다. 또한 신경망 학습의 목적은 신경망의 출력(Softmax의 출력)이 정답 레이블과 가까워지도록 가중치 매개변수의 값을 조정하는 것인데, 신경망의 출력과 정답 레이블의 오차를 앞 계층에 전달하는 것이 있는 그대로의 차이를 전달하는 것이라 효율적이다.

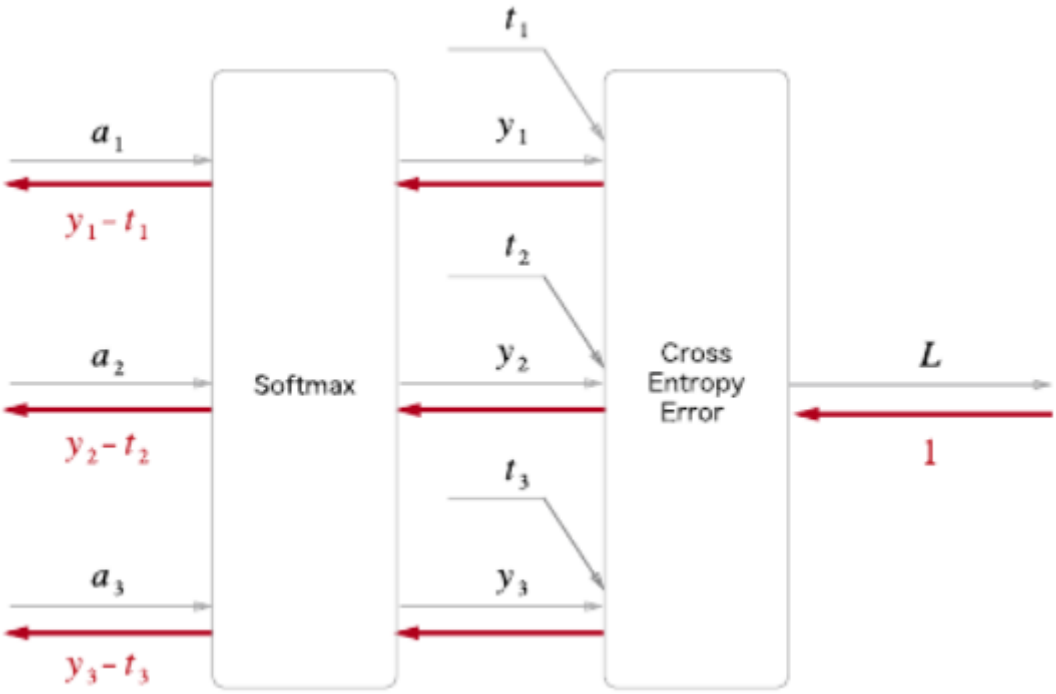
신경망의 출력이 정답 레이블과 비슷할수록 Softmax 계층의 역전파가 보내는 오차가 작아진다. (학습하는 정도가 작아진다.)

Softmax-with-Loss 계층 구현

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None
        self.t = None

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)
        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t) / batch_size # 역전파 시 값을 배치의 수로 나누서 데이터 1개당 오차를 앞 계층으로 전파한다.

        return dx
```

이 구현에서는 소프트맥스 함수 구현시 주의점(3.5.2)과 배치용 교차 엔트로피 오차 구현하기(4.2.4)에서 구현한 함수인 softmax()와 cross_entropy_error()를 이용했다.

부록 A: Softmax-with-Loss 계층의 계산 그래프

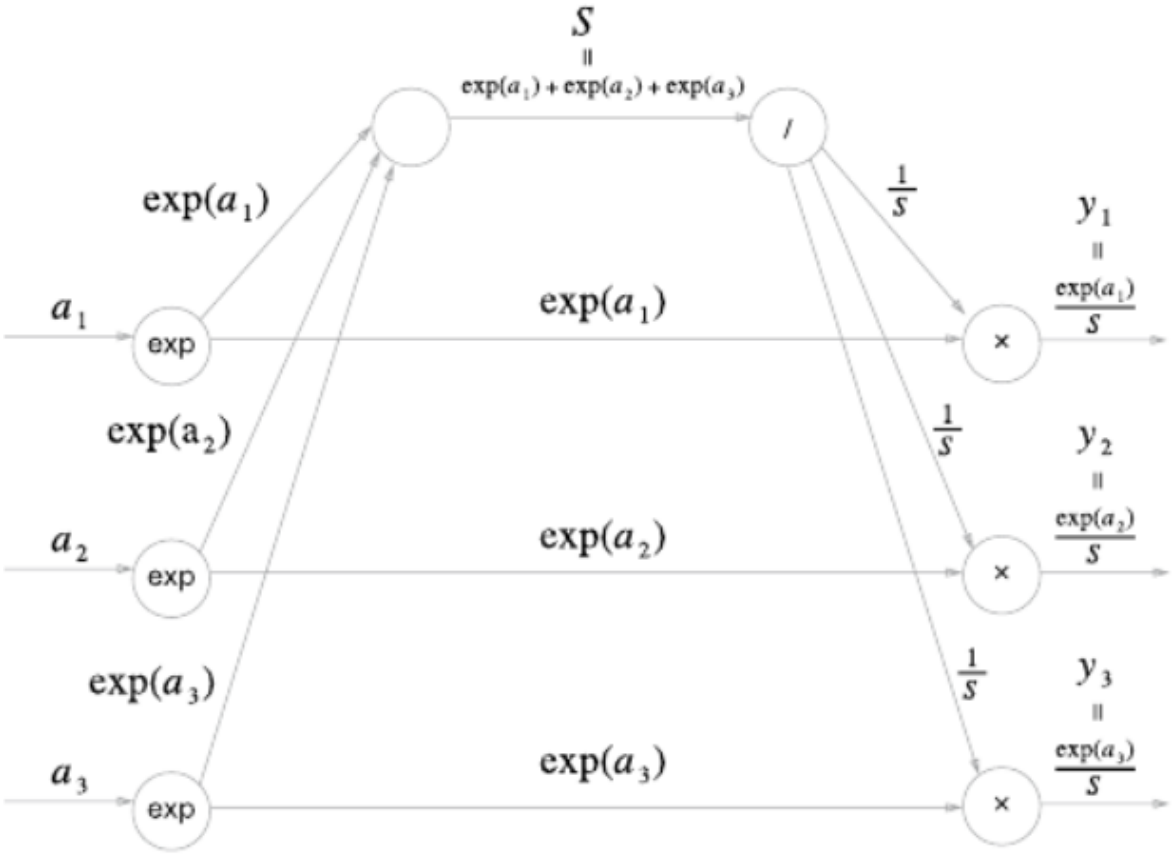


3분류를 수행한다. 이전 계층으로부터의 입력은 (a1, a2, a3) 이며 Softmax 계층은 (y1, y2, y3) 를 출력한다. 정답 레이블은 (t1, t2, t3) 이며 Cross Entropy Error계층은 손실 L을 출력한다. 역전파 결과는 (y1 - t1, y2 - t2, y3 - t3) 이다.

A.1 순전파

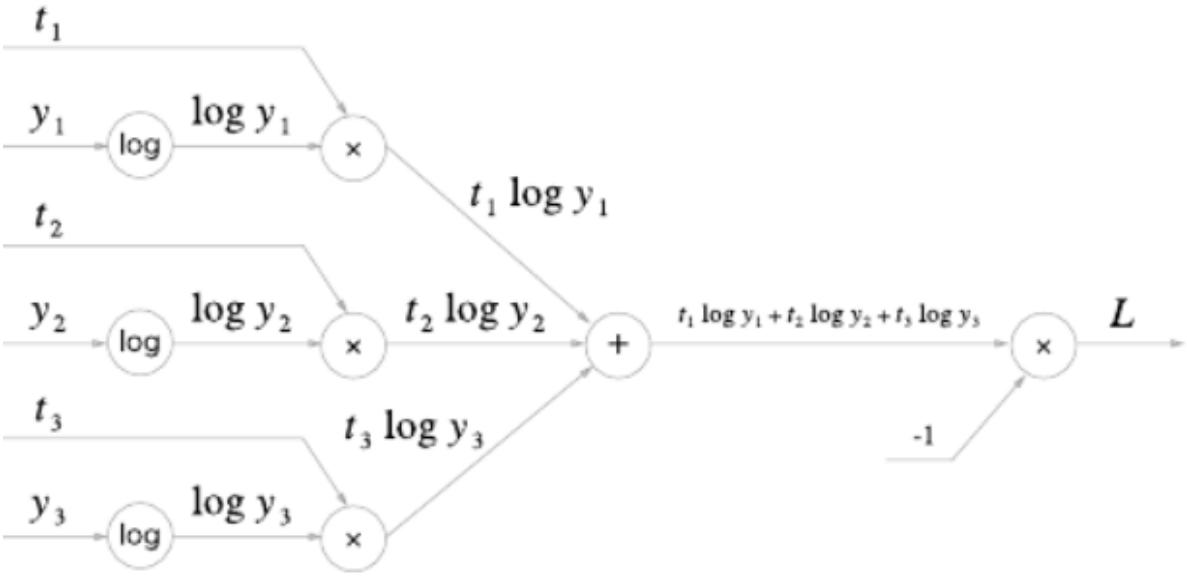
소프트 맥스 함수: $$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

소프트 맥스 계층 순전파:



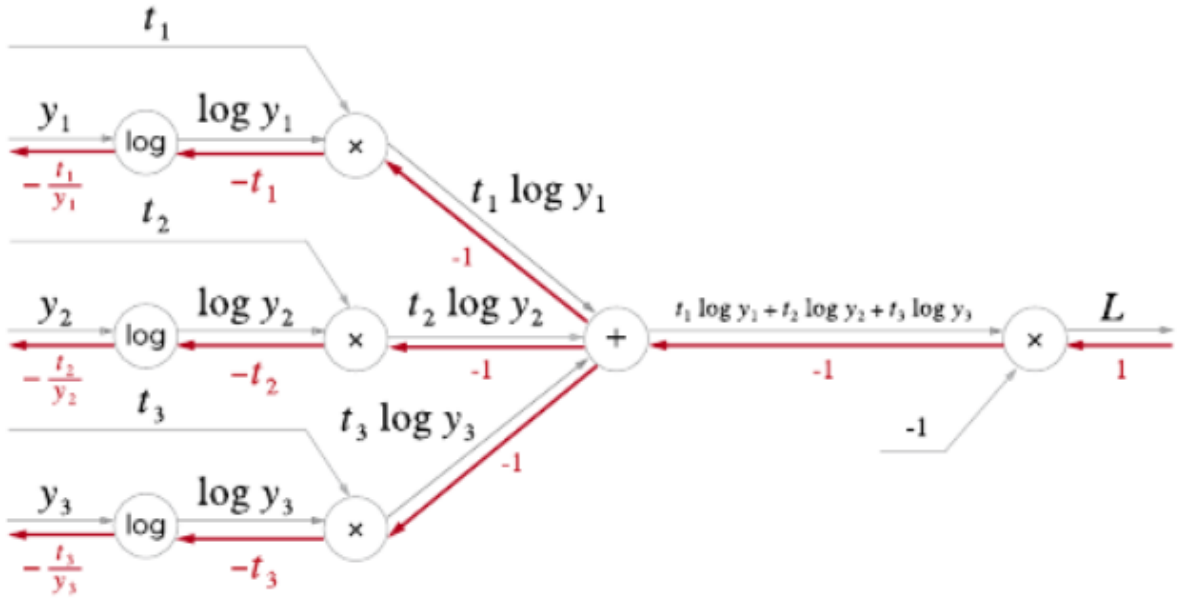
교차 엔트로피 오차 수식: $$L = -\sum_k t_k \log y_k$$

교차 엔트로피 오차 계층 순전파:



A.2 역전파

교차 엔트로피 오차 계층의 역전파:



- 역전파의 초깃값, 즉 가장 오른쪽 역전파의 값은 1이다 (dL/dL=1)
- `x` 노드의 역전파는 순전파 시 입력들의 값을 서로 바꿔 상류의 미분에 곱하고 하류로 흘린다.
- `+` 노드에서는 상류에서 전해지는 미분을 그대로 흘린다.
- `log` 노드의 역전파는 다음 식을 따른다.

$$y = \log x$$

  - $$\frac{\partial y}{\partial x} = \frac{1}{x}$$

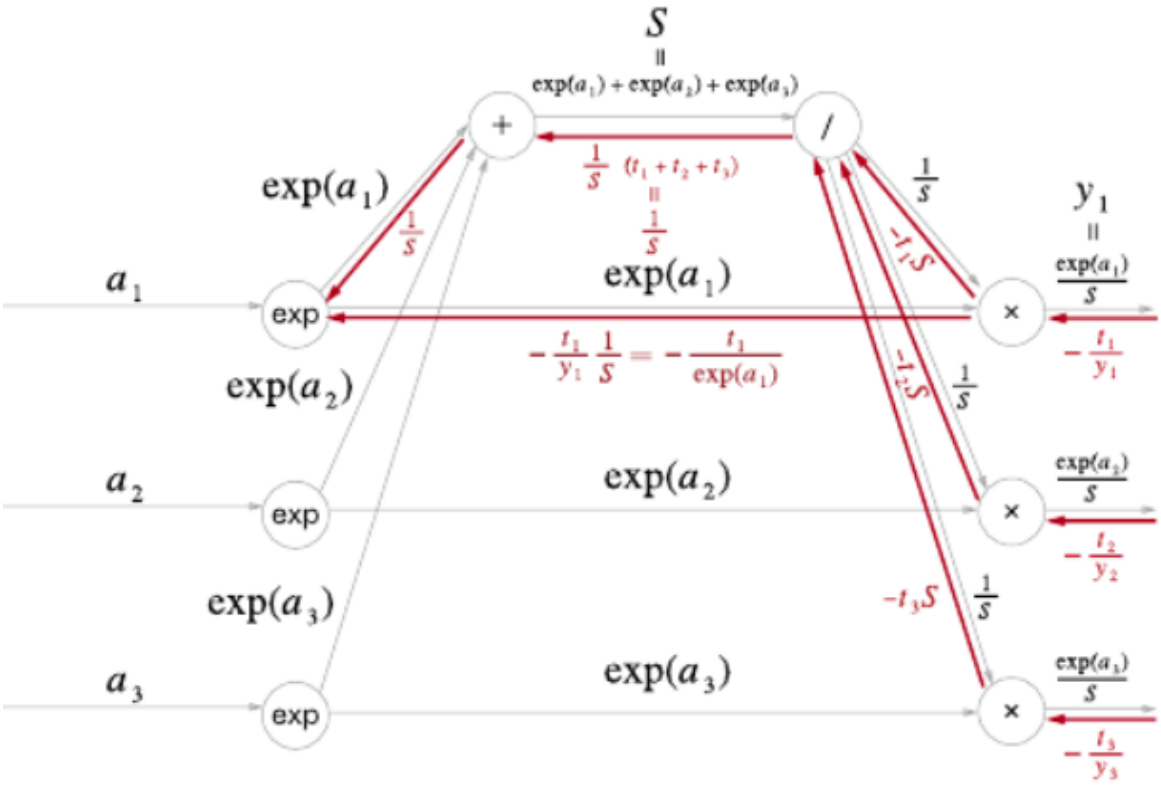
교차 엔트로피 오차 계층의 역전파의 결과는 (-(t1/y1) , -(t2/y2), -(t3/y3))이며, 이 값이 Softmax 계층 역전파의 입력이 된다.

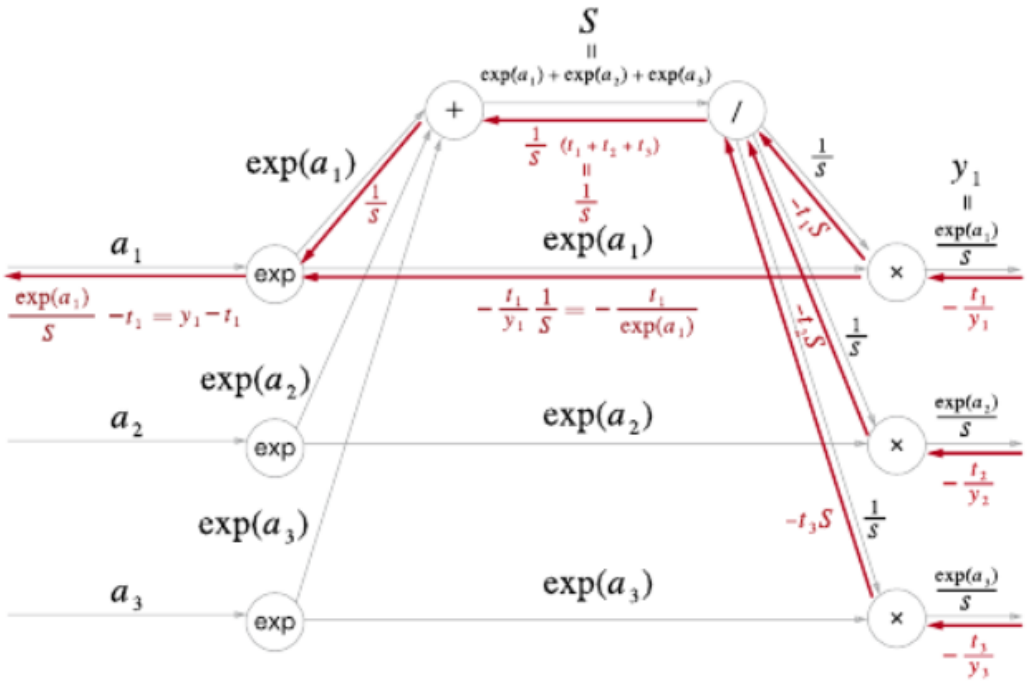
Softmax 계층 역전파 :

- 1단계: 앞 계층 (교차 엔트로피 오차 계층)의 역전파 값이 흘러온다.
- 2단계: 순전파의 입력들을 서로 바꿔 곱한다.

$$-\frac{t_1}{y_1}\exp(a_1) = -t_1\frac{S}{\exp(a_1)}\exp(a_1) = -t_1 S$$

- 3단계: 순전파 때 여러 갈래로 나뉘어 흘렸다면 역전파 때는 그 반대로 흘러온 여러 값을 더한다. (-(t1 x S) -(t2 x S) - (t3 x S)). 이 더해진 값이 `/` 노드의 역전파를 거쳐 (t1 + t2 + t3) / S 이 된다. 그런데 여기서 (t1, t2, t3)은 one-hot 벡터 이므로 t1 + t2 + t3 = 1 이 된다.
- 4단계: `+` 노드는 그냥 흘려보내고
- 5단계: `x` 노드는 입력을 서로 바꾼 곱셈을 한다.



- 6단계: `exp` 노드에서는 다음 관계식이 성립된다.
  - $$y = \exp(x)$$
  - $$\frac{\partial y}{\partial x} = \exp(x)$$
  - 두 갈래의 입력의 합에 exp(a1) 을 곱한 수치가 여기에서 구하는 역전파이다.
  - 

  - 최종 역전파는 y1 - t1 이다.

5.7 오차역전파법 구현하기

#신경망구축

구현했던 계층을 조합하면 신경망을 구축할 수 있다.

5.7.1 신경망 학습의 전체 그림

- 전제
  - 신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다.
- 1단계: 미니배치
  - 훈련 데이터 중 일부를 가져온다. 이 미니배치의 손실 함수 값을 줄이는 것이 목표이다.
- 2단계: 기울기 산출
  - 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시한다. (오차역전파법)
- 3단계: 매개변수 갱신
  - 가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다. (학습률)
- 4단계: 반복
  - 1 ~ 3단계를 반복한다.

5.7.2 오차역전파법을 적용한 신경망 구현하기

2층 신경망을 구현한다. 계층을 사용함으로써 인식 결과를 얻는 처리(predict())와 기울기를 구하는 처리(gradient()) 계층의 전파만으로 동작이 이루어진다.

표 5-1 TwoLayerNet 클래스의 인스턴스 변수

인스턴스 변수	설명
params	딕셔너리 변수로, 신경망의 매개변수를 보관
	params['W1']은 1번째 층의 가중치, params['b1']은 1번째 층의 편향
	params['W2']는 2번째 층의 가중치, params['b2']는 2번째 층의 편향
layers	순서가 있는 딕셔너리 변수로, 신경망의 계층을 보관
	layers['Affine1'], layers['Relu1'], layers['Affine2']와 같이 각 계층을 순서대로 유지
lastLayer	신경망의 마지막 계층
	이 예에서는 SoftmaxWithLoss 계층

표 5-2 TwoLayerNet 클래스의 메서드

메서드	설명
__init__(self, input_size, hidden_size, output_size, weight_init_std)	초기화를 수행한다.
	인수는 앞에서부터 입력층 뉴런 수, 은닉층 뉴런 수, 출력층 뉴런 수, 가중치 초기화 시 정규분포의 스케일
predict(self, x)	예측(추론)을 수행한다.
	인수 x는 이미지 데이터
loss(self, x, t)	손실 함수의 값을 구한다.
	인수 x는 이미지 데이터, t는 정답 레이블
accuracy(self, x, t)	정확도를 구한다.
numerical_gradient(self, x, t)	가중치 매개변수의 기울기를 수치 미분 방식으로 구한다(앞 장과 같음).
gradient(self, x, t)	가중치 매개변수의 기울기를 오차역전파법으로 구한다.

```
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict

class TwoLayerNet:

    def __init__(self, input_size, hidden_size,
                 output_size, weight_init_std = 0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size,
hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size,
output_size)
        self.params['b2'] = np.zeros(output_size)

        # 계층 생성
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
```

```
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])

        self.lastLayer = SoftmaxWithLoss()

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(sself, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)  #one-hot처리

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        # 순전파
        self.loss(x, t)

        # 역전파
        dout = 1
        dout = self.lastLayer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 결과 저장
        grads = {}
        grads['W1'] = self.layers['Affine1'].dW
        grads['b1'] = self.layers['Affine1'].db
        grads['W2'] = self.layers['Affine2'].dW
        grads['b2'] = self.layers['Affine2'].db

        return grads
```

신경망의 계층을 OrderedDict에 보관하는 점이 중요하다. (https://www.daleseo.com/python-collections-ordered-dict/)

OrderedDict을 사용함으로써 순전파 때는 추가한 순서대로 각 계층의 forward() 메서드를 호출하기만 하면 된다. 역전파 때는 계층을 반대 순서로 호출하기만 하면 된다.

5.7.3 오차역전파법으로 구한 기울기 검증하기

기울기를 구하는 방법

- 수치 미분
- 해석적으로 수식을 풀어 구하는 방법

해석적 방법은 오차역전파법을 이용하여 매개변수가 많아도 효율적으로 계산할 수 있었다. 느린 수치미분 대신 오차 역전파법을 사용해보자.

수치 미분과 오차역전파법의 결과를 비교하여 오차역전파법을 검증한다. 두 방식으로 구한 기울기가 일치함(거의 같음)을 확인하는 작업을 `기울기 확인 gradient check` 라고 한다.

```
# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True,
one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

x_batch = x_train[:3]
t_batch = t_train[:3]

grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)

# 각 가중치의 차이의 절댓값을 구한 후, 그 절댓값들의 평균을 낸다.
for key in grad_numerical.keys():
    diff = np.average(np.abs(grad_backprop[key] - grad_numerical[key]))
    print(key + ":" + str(diff))
```

오차가 0에 가까운 값들이 나온다. 만약 오차가 크다면 오차역전파법을 잘못 구현했다고 의심해봐야 한다.

5.7.4 오차역전파법을 사용한 학습 구현하기

```
# 데이터 읽기
(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True)
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼 파라메터
iters_num = 10000  # 반복횟수
train_size = x_train.shape[0]
batch_size = 100  # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
```

```
train_acc_list = []
test_acc_list = []

# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    # print(i)
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 오차역전파법으로 기울기 계산
    grad = network.gradient(x_batch, t_batch)

    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    # 1에폭 당 정확도 계산
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# 결과 :
# train acc, test acc | 0.11421666666666666, 0.1154
# train acc, test acc | 0.9036666666666666, 0.9054
# train acc, test acc | 0.9232166666666667, 0.926
# train acc, test acc | 0.93555, 0.9365
# train acc, test acc | 0.9426333333333333, 0.9426
# train acc, test acc | 0.9494333333333334, 0.9482
# train acc, test acc | 0.9550833333333333, 0.954
# train acc, test acc | 0.95925, 0.9575
# train acc, test acc | 0.9633833333333334, 0.9604
```

기울기를 수치 미분이 아닌 오차역전파법으로 구한다.

5.8 정리

계산 그래프를 이용하여 신경망의 동작과 오차역전파법을 설명하고, 그 처리 과정을 계층이라는 단위로 구현하였다.

모든 계층에서 forward와 backward라는 메서드를 구현한다. 매개변수의 기울기를 효율적으로 구할 수 있다.

동작을 계층으로 모듈화한 덕분에 계층을 조합하여 원하는 신경망을 쉽게 만들 수 있다.